BUSINESS INTELLIGENCE AND DATA ANALYTICS

University of Petra

Institute of Analytics Recognized Programme

University of Petra

Introduction to Word Embeddings

307307 BI Methods

Converting Words to Numbers

Old Method:- One-Hot Encoding
Each word is represented by setting one dimension to 1 and all the other dimensions to ZEROS.

id	color
1	red
2	blue
3	green
4	blue

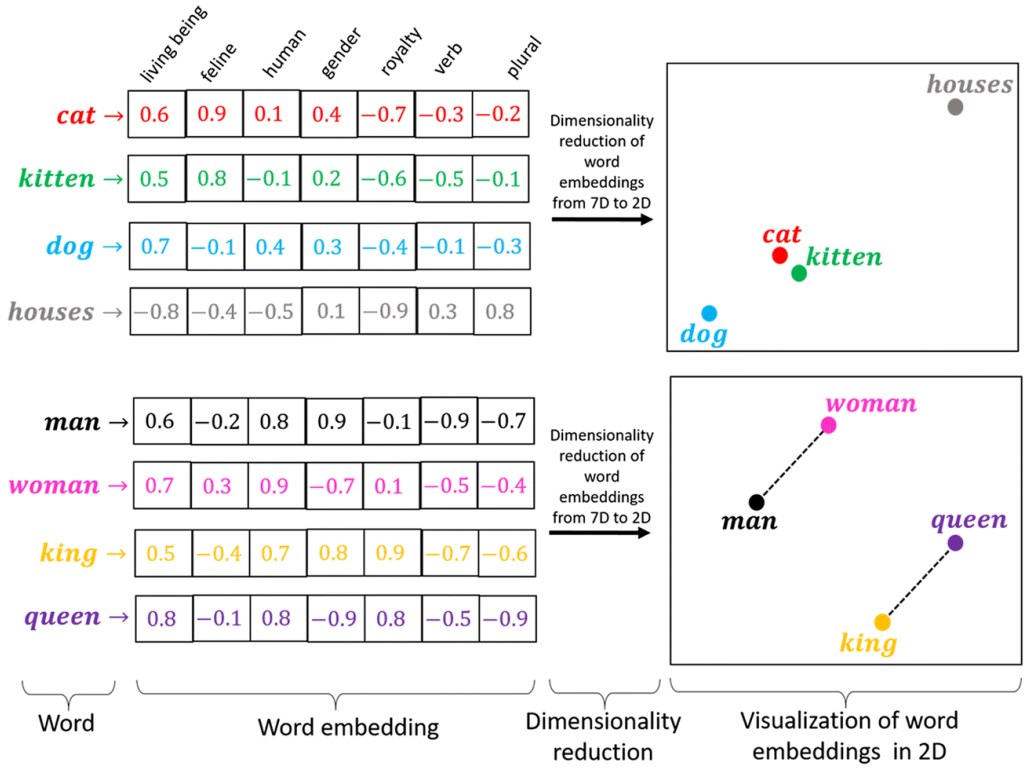
id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0

Limitations:

- Vocabulary of 100,000 words → 100,000 dimensions [0010000000000000000000000000000000000000...etc.]
- All words equally distant from each other
- No semantic meaning captured

Word Embeddings and Vector Spaces

- Words are encoded as dense numerical vectors instead of one-hot or sparse representations.
- Similar words → similar vectors
- Typically, 50-300 dimensions (vs. vocabulary size in 1-hot-encoding)
- Word Embeddings captures semantic and syntactic relationships about/between words.
- Each dimension potentially captures some syntactic or semantic meaning.
- These vectors are learned from text by models like Word2Vec.

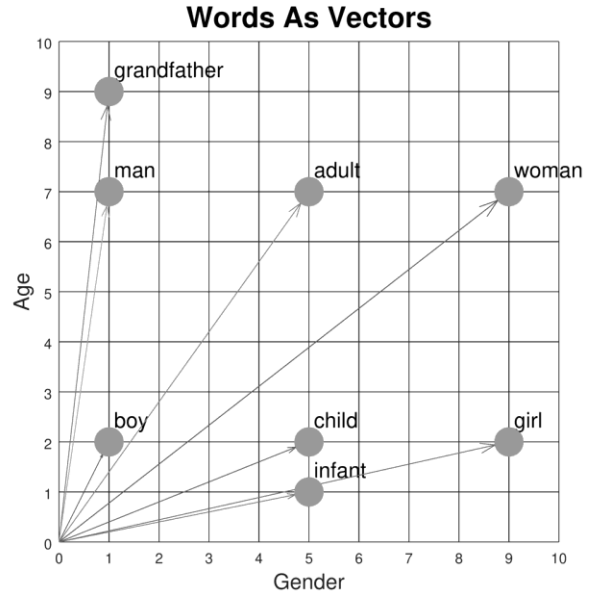


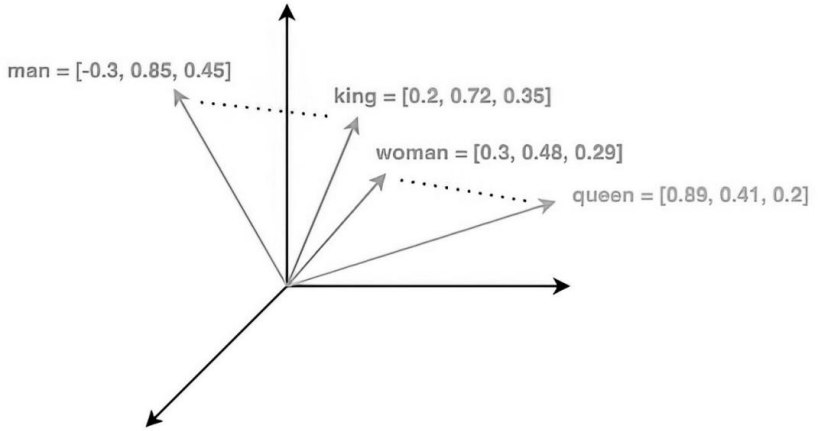
Vector Spaces and Word Embeddings

- A vector space is a mathematical space where each word is represented as a point (or vector) in multi-dimensional space.
- Makes it possible to compare, visualize, and manipulate meanings of words using math.
- Enables operations like:
  - Similarity: "king" is close to "queen"
  - Analogy: "king" - "man" + "woman" ≈ "queen"

Properties of Vector Space

- Semantic relationships are preserved (e.g., “man" is closer to “grandfather" than “girl").
- Closer Vectors → Similar Meanings
- Vectors Farther Apart → Dissimilar Meanings





Key Intuition

- Distributional hypothesis - "You shall know a word by the company it keeps" (J.R. Firth, 1957)
- Words in similar contexts have similar meanings:
- "The **cat** sat on the mat"
- "The **dog** sat on the mat"
- "The **rabbit** sat on the mat"
- → cat, dog, rabbit learn similar representations

Word2Vec (Tomas Mikolov et al., 2013)

Word2Vec was developed by Tomas Mikolov and team at Google.

Mikolov presented two architectures:

1) CBOW (Continuous Bag of Words): Predict word from context

- Input: [the, ___, sat, on] → Output: cat

2) Skip-gram: Predict context from word

- Input: cat → Output: [the, sat, on, mat]

- Trained on massive text corpora using simple neural networks

Mikolov Approach – Convert Text into Input-Output Pairs

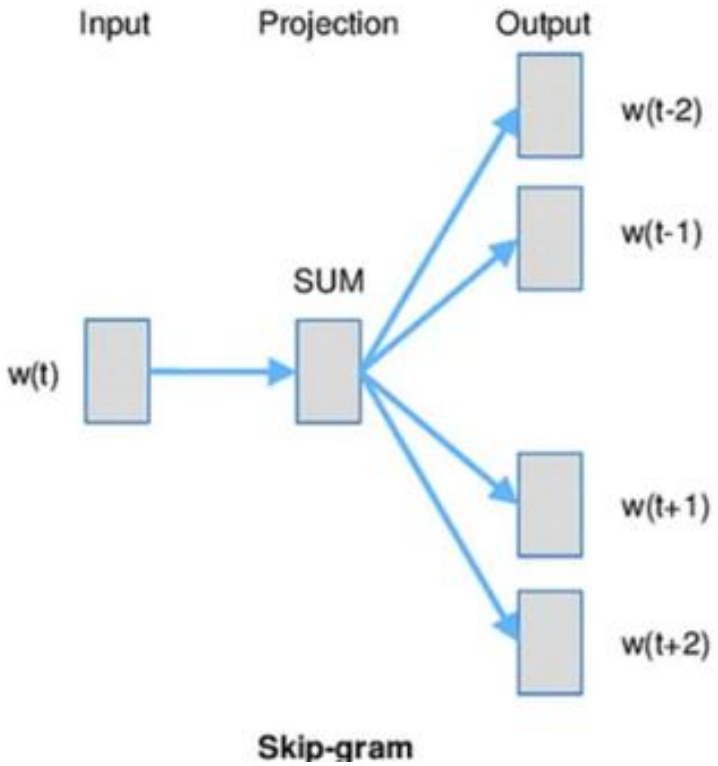
Suppose we have these sentences (corpus), we can convert them into input output pairs to be used for training a neural network:

Large language models are transforming business applications by introducing new levels of automation, intelligence, and personalization across industries. These models, trained on vast amounts of text data, can understand and generate human-like language, allowing businesses to enhance customer interactions, streamline operations, and gain deeper insights from data. In customer service, for instance, large language models power advanced chatbots and virtual assistants that can handle complex queries, provide 24/7 support, and adapt to customer needs in real time. In marketing, they help craft personalized messages, analyze consumer sentiment, and optimize content strategies with remarkable precision. Organizations are also using these models to automate report generation, summarize large documents, and even assist in writing code or drafting legal documents, significantly reducing time and cost. Moreover, they enable better decision-making by turning unstructured data—such as emails, reviews, or social media posts—into actionable business intelligence. As companies integrate these tools, they not only boost productivity but also redefine how humans and machines collaborate in the workplace. However, this transformation also raises questions about data privacy, ethical AI use, and workforce adaptation, making responsible deployment as important as technological innovation itself.

Skip-gram Architecture

- **Source Sentence:** "Large language models are transforming business applications"
- **Window Size:** 2 (2 words on each side of target word)
- **Task:** Predict context words from target word
- Given Target Word Predict → Context Words

Input (Target Word)	Output (Context Words)
Large	[language, models]
language	[Large, models, are]
models	[Large, language, are, transforming]
are	[language, models, transforming, business]
transforming	[models, are, business, applications]
business	[are, transforming, applications]
applications	[transforming, business]

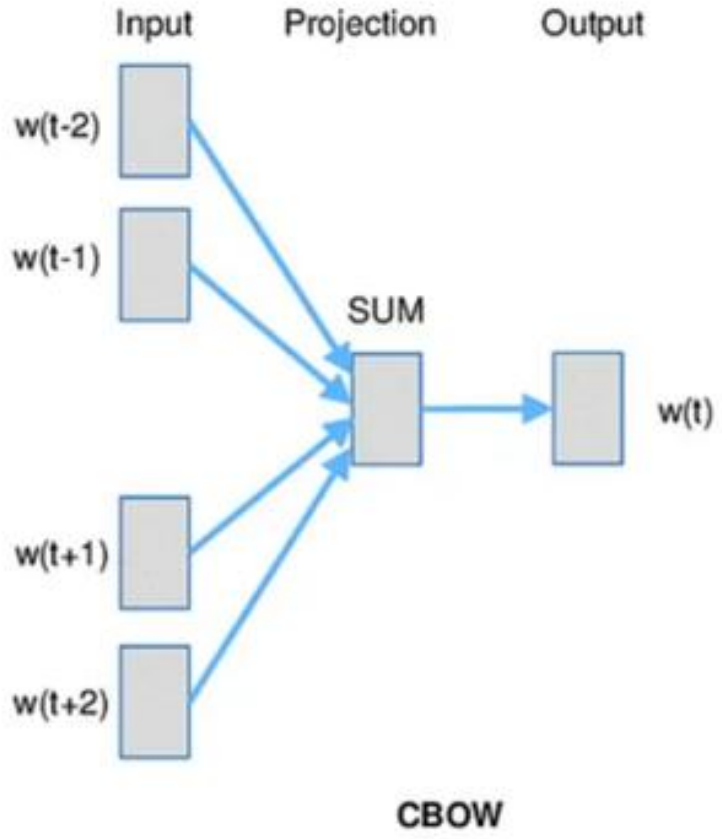


Skip-gram

CBOW (Continuous Bag of Words) Architecture

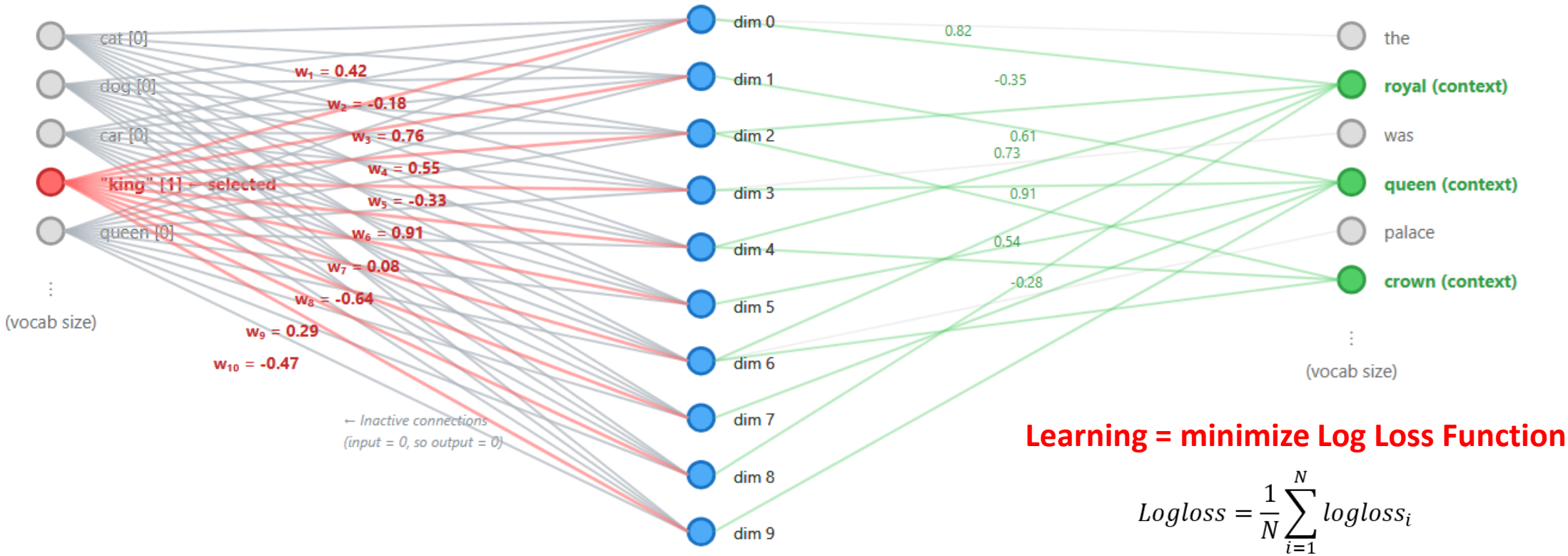
- **Source Sentence:** "Large language models are transforming business applications"
- **Task:** Predict target word from context words
- Given Context Words Predict → Target Word

Input (Context Words)	Output (Target Word)
[language, models]	Large
[Large, models, are]	language
[Large, language, are, transforming]	models
[language, models, transforming, business]	are
[models, are, business, applications]	transforming
[are, transforming, applications]	business
[transforming, business]	applications



Word2Vec Skip-gram: Learning Word Embeddings

Training sentence: "The **king** wore a royal crown, and the queen stood beside him"



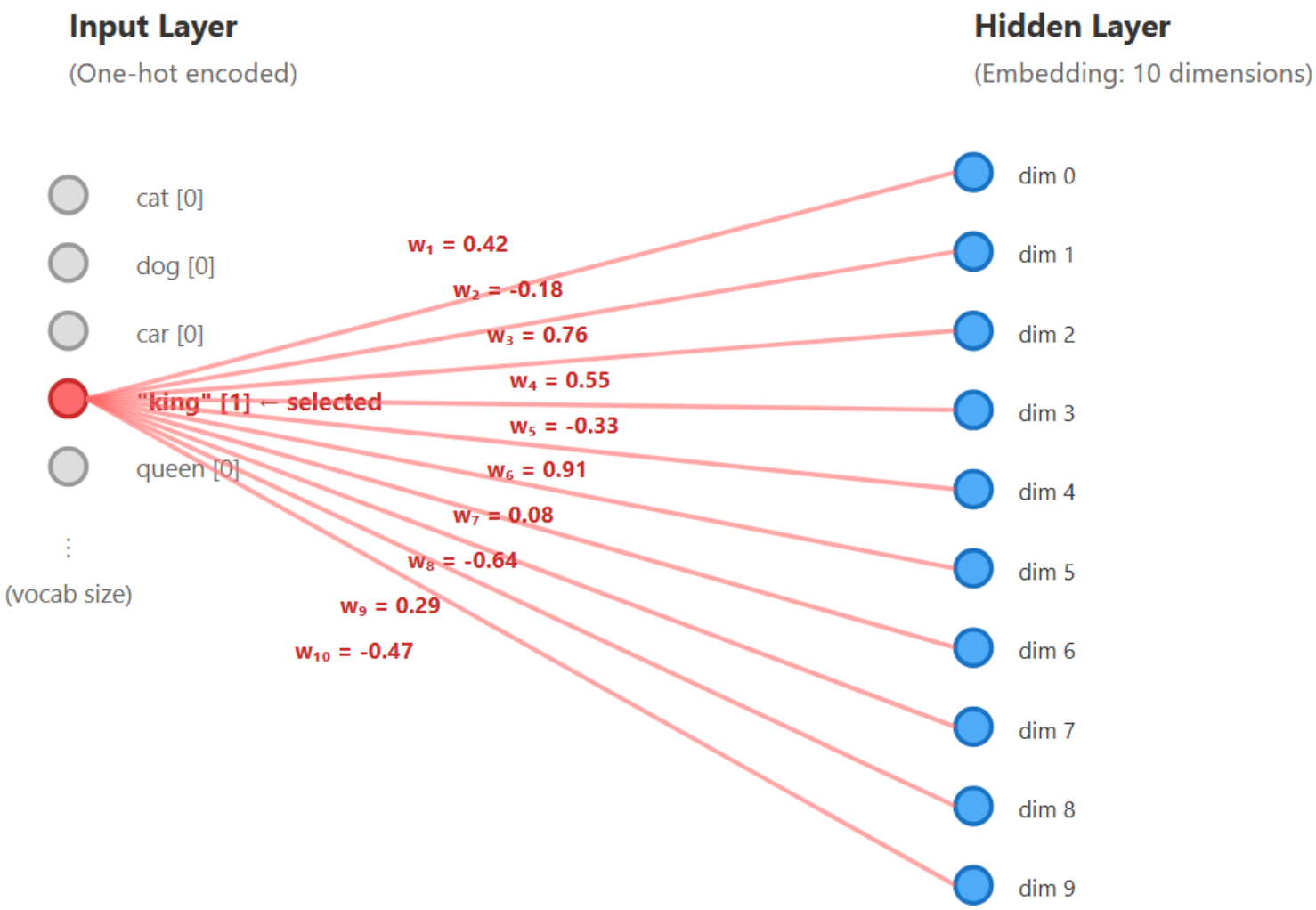
Learning = minimize Log Loss Function

$$Logloss = \frac{1}{N}\sum_{i=1}^{N} logloss_i$$

Training Process: Input Word "king" → Embedding → Predict Context Words

- Input→Hidden weights = Word Embeddings (e.g., king = [0.42, -0.18, 0.76, 0.55, -0.33, 0.91, 0.08, -0.64, 0.29, -0.47])
- Hidden→Output weights predict context: Given "king", maximize probability of "royal", "queen", "crown"
- Backpropagation updates both weight matrices → words in similar contexts get similar embeddings

Skip-Gram Model



Core Math Behind Word2Vec: Softmax & Log-Loss

- **Softmax:** converts raw scores into probabilities

$$P_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Example: suppose the model gives raw scores $[2, 1, 0.1]$. Softmax turns these into probabilities $[0.66, 0.24, 0.10]$, so the highest score becomes the highest probability, and all probabilities sum to 1.

- **Log loss (or cross-entropy loss):** measures how well the predicted probabilities match the correct answer

$$L = -\log(P_{\text{correct}})$$

Example:

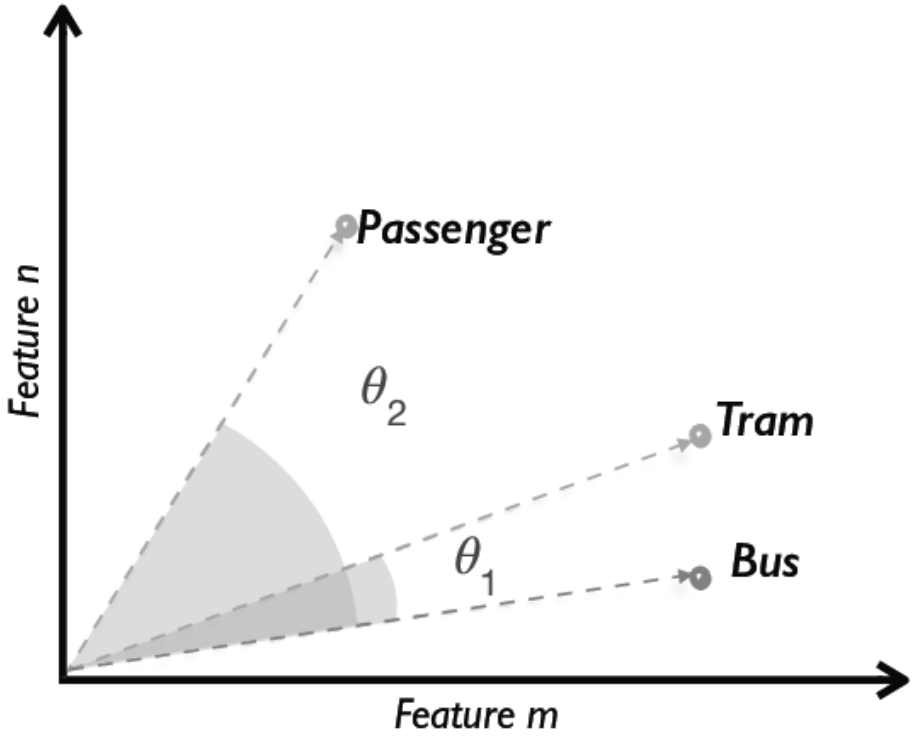
- If $P_{\text{correct}} = 0.9$, then $L = -\log(0.9) \approx 0.11$
- If $P_{\text{correct}} = 0.1$, then $L = -\log(0.1) \approx 2.30$

Higher probability for the correct word gives a smaller loss, and lower probability gives a larger loss.

Measuring Similarity Between Word Vectors

Why Compare Word Vectors?

- Word embeddings map words into a vector space.
- **Words with similar meanings** are placed **close together** in that space.
- To quantify this "closeness," we use **vector similarity**.



Cosine Similarity

Most common metric used to compare word vectors:

$$\text{cosine\_similarity}(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\|\|\vec{B}\|}$$

- Measures the **angle** between two vectors (not their magnitude).
- Ranges from **-1 to 1**:
  - 1 → Same direction (very similar)
  - 0 → Orthogonal (unrelated)
  - -1 → Opposite directions (very different)

Intuition

- Vectors for `"king"` and `"queen"` will have high cosine similarity.
- Vectors for `"apple"` and `"keyboard"` will have low similarity.

Use Pre-Trained Embeddings

Gensim is an open-source Python library used for **topic modeling** and **natural language processing (NLP)**.
It's great for working with **large text datasets** because it doesn't need to load all the data into memory at once.

Key features:

- Builds and uses **word embeddings** (like Word2Vec, FastText, Doc2Vec).
- Measures **semantic similarity** between words or documents.
- **Efficient and memory-friendly**, ideal for handling big collections of text.

```python
import gensim.downloader as api
from gensim.models import Word2Vec

# Load pre-trained Word2Vec model
word2vec_model = api.load("word2vec-google-news-300")


# Get vector for a word
cat_vector = model.wv['cat']
print("Vector for 'cat':", cat_vector[:5])  # Show first
5 dimensions

# Find similar words
similar_words = word2vec_model.most_similar('computer',
   topn=5)
print("Words similar to 'computer':", similar_words)

# Word analogies
result = word2vec_model.most_similar(positive=['woman',
   'king'], negative=['man'], topn=1)
print("king - man + woman =", result)
```

Open in Colab

Applications of Word Embeddings in NLP

1. **Semantic Similarity**
   Measure how similar two words, phrases, or documents are by comparing their vector representations.
   Example: Identifying that "doctor" and "physician" are closely related.

2. **Text Classification**
   Used as input features for tasks like spam detection, sentiment analysis, and topic classification.
   Embeddings provide rich, dense input for machine learning models.

3. **Named Entity Recognition (NER)**
   Help identify proper nouns and classify them into categories like person, location, or organization.
   Embedding-based models improve contextual understanding of named entities.

4. **Machine Translation**
   Map words from one language to another by aligning embeddings in multilingual space.
   Improves translation accuracy by leveraging semantic proximity.

5. **Question Answering & Chatbots**
   Used to understand queries and match them with appropriate answers or responses.
   Enable bots to interpret intent and context more accurately.

- **6. Information Retrieval**
  Enhance search engines by retrieving results based on semantic meaning, not just keyword matches.
  Example: Searching for “heart attack” returns documents containing “cardiac arrest.”

Context Aware Embeddings

Contextual Word Embeddings (BERT and GPT)

Key Innovation

Unlike static embeddings (Word2Vec, GloVe), contextual models generate **different vectors for the same word** depending on its context.

Approach

- Uses deep, pre-trained neural networks (often transformer-based)
- Embeddings are derived from entire sentences, capturing syntax and semantics dynamically

Examples

- **BERT - Bidirectional Encoder Representations from Transformers (2018)**: Transformer-based neural networks trained with masked language modeling and next sentence prediction
- **GPT - Generative Pre-training Transformer (2018):** Transformer-based unidirectional language model focused on generation.

Characteristics

- Embeddings are **context-sensitive** (e.g., “bank” in “river bank” vs. “savings bank”)
- Each word is embedded based on its role in the sentence.
- Embeddings vary for the same word depending on its position and meaning.
- Significantly improve performance on downstream NLP tasks.

BERT: Bidirectional Encoder Representations from Transformers

- Developed by Google in 2018
- A **pre-trained language model** based on the **Transformer encoder**
- Reads text **bidirectionally**, enabling deep contextual understanding

Key Ideas

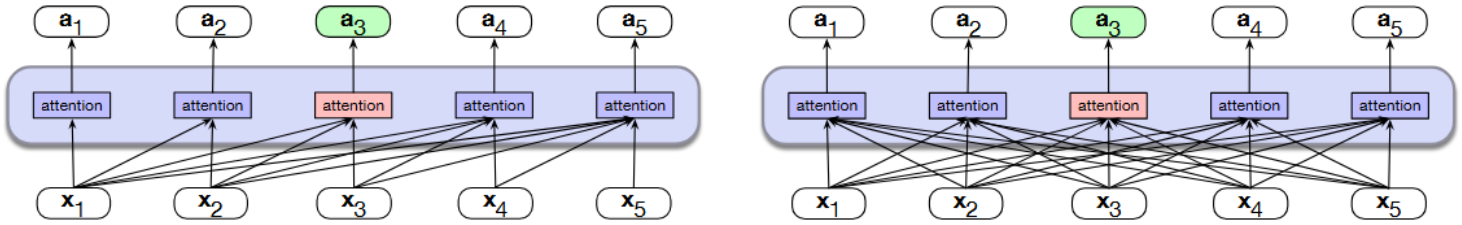
- Uses only the **encoder** stack of the Transformer
- Pre-trained on large text corpora, then fine-tuned on specific tasks

Pretraining Objectives

- **Masked Language Modeling (MLM)**: Predict randomly masked words in a sentence
- **Next Sentence Prediction (NSP)**: Predict if one sentence follows another

Applications

- Sentiment Analysis
- Question Answering
- Named Entity Recognition
- Text Classification
- Semantic Search



a) A causal self-attention layer

b) A bidirectional self-attention layer

More About BERT

Key Advantages

- Contextual embeddings: Word meanings change based on context
- Captures long-range dependencies
- Pre-trained on massive datasets → Transfer learning
- State-of-the-art performance on 11 NLP tasks when released

How BERT Works:

- **Multiple stacked encoder layers** with self-attention mechanisms
- **No decoders** - purely focused on understanding input
- **Captures relationships** between words and their context
- **Powerful context learning** - understands word relationships in sentences
- **Meaningful representations** - transforms text into useful numbers

Architecture Overview

```
Input: [CLS] The cat sat on the mat [SEP]
          ↓
    Token Embeddings
    Position Embeddings
    Segment Embeddings
          ↓
    Transformer Encoder (12 or 24 layers)
          ↓
    Contextual Representations
```

Key Components

- **[CLS]**: Classification token (sentence-level tasks)
- **[SEP]**: Separator token (between sentences)
- **Multi-head attention**: Allows model to focus on different positions
- **Feed-forward networks**: Process attention outputs

BERT's Training Data

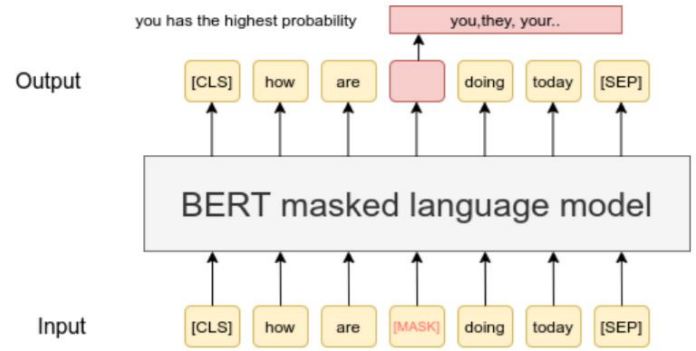
Pre-training Foundation

- **Training corpus**: 3+ billion words
  - English Wikipedia
  - ~10,000 unpublished books
- **Clean, large-scale dataset** for comprehensive language learning
- **Pre-training approach** to learn language patterns and context

Training Objective 1 - Masked Language Modeling

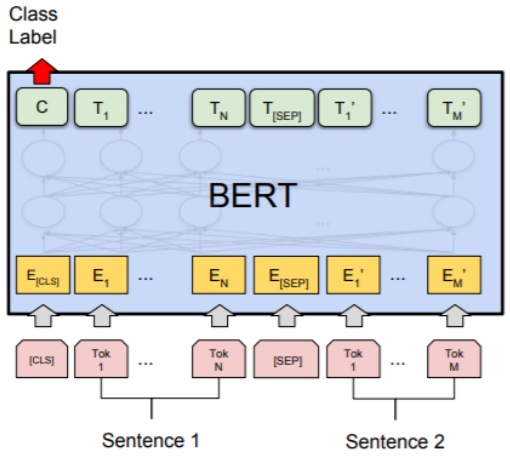
Learning Through "Fill in the Blanks"

- **Random word masking**: Some words are hidden during training
- **Context-based prediction**: BERT predicts masked words using surrounding context
- **Example**: "I love eating _____ in my fruit salad" → "lemons"
- **Key benefit**: Learns word relationships and contextual understanding
- **Powerful concept**: Used in many other AI applications



Training Objective 2 - Next Sentence Prediction

- **Title: Understanding Sentence Relationships**
- **Sentence pair analysis**: Given two sentences, determine if B follows A
- **Logical connection assessment**: Analyzes context and content
- **Example**:
  - A: "The cat climbed the tree"
  - B: "It was trying to catch a bird" ✓ (follows logically)
  - C: "The weather is nice today" ✗ (unrelated)
- **Note**: Later removed in newer models (MLM proved sufficient)



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

BERT's Ideal Use Cases

Where BERT Excels

- **Text Classification**: Sentiment analysis, topic classification, spam detection
- **Named Entity Recognition**: Identifying people, organizations, locations, dates
- **Extractive Question Answering**: Finding answers within provided context
- **Semantic Similarity**: Measuring similarity between sentences/paragraphs
- **Key advantage**: Perfect for understanding tasks, not generation

Extractive Question Answering Example

How BERT Answers Questions:

- **Process**: Analyzes both context and question
- **Method**: Identifies start and end tokens of the answer
- **Example**:
  - Context: "Mount Everest is the highest mountain..."
  - Question: "What is the highest mountain?"
  - Answer: "Mount Everest" (extracted, not generated)
- **Important**: BERT extracts existing text, doesn't generate new content

Component	BERT-Base (L=12, H=768, A=12)	BERT-Large (L=24, H=1024, A=16)
Embedding Layer - Token + Positional + Segment	30522 × 768 + 512 × 768 + 2 × 768 ≈ 23.8M	30522 × 1024 + 512 × 1024 + 2 × 1024≈ 31.4M
Self-Attention - Q/K/V + Output per layer	$4 \times H^2 = 4 \times 768^2 = 2.36M$	$4 \times 1024^2 = 4.19M$
Total across all layers	12 × 2.36M = 28.3M	24 × 4.19M = 100.6M
Feedforward - Two linear layers per layer	768 × 3072 + 3072 × 768 = 4.71M	1024 × 4096 × 2 = 8.39M
Total across all layers	12 × 4.71M = 56.5M	24 × 8.39M = 201.4M
LayerNorms - Two per layer	2 × 768 = 1.5K × 12 = 18K	2 × 1024 × 24 = 49K
Pooler - Final CLS output to 768 or 1024	768 × 768 = 0.59M	1024 × 1024 = 1.05M
Total Parameters	**≈ 110M**	**≈ 340M**

- **L: number of layers (Transformer blocks)**
- **H: hidden size**
- **A: number of attention heads (H / A = size per head)**
- **Vocabulary size: 30,522**
- **FFN hidden size: 4×H (3072 for base, 4096 for large)**

BERT Variants

Common BERT Models

Model	Parameters	Layers	Hidden Size	Attention Heads
BERT-Base	110M	12	768	12
BERT-Large	340M	24	1024	16
DistilBERT	66M	6	768	12
RoBERTa	355M	24	1024	16
ALBERT	12M-235M	12-24	768-4096	12-64

Specialized Variants

- **BioBERT**: Biomedical text
- **SciBERT**: Scientific text
- **FinBERT**: Financial text
- **ClinicalBERT**: Clinical notes

Example: Print out BERT Embeddings

```
from transformers import BertTokenizer, BertModel
import torch

# Load pretrained BERT
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased’)

# Sentence
sentence = "He went to the bank to deposit money.“

# Tokenize
inputs = tokenizer(sentence, return_tensors='pt’)

# Get outputs
with torch.no_grad():  # No training, just inference
    outputs = model(**inputs)

# Get the hidden states (embeddings)
embeddings = outputs.last_hidden_state  # Shape: (batch_size, sequence_length,

# (hidden_size)
print(embeddings.shape)  # Example output: torch.Size([1, 11, 768])
```

Open in Colab

GPT – Overview and Architecture

What is Generative **P**re-trained **T**ransformer (GPT)?

- A family of **Transformer-based language models** developed by OpenAI
- Uses only the **decoder stack** of the original Transformer architecture
- Trained with **causal (autoregressive) language modeling** to predict the next token
- Focuses on generating human-like text (unlike BERT's understanding focus)

Training Objective

- Predict the next token in a sequence

GPT Variants

- **GPT-1 (2018, 120M Parameters)**: Introduced the pretrain-then-finetune paradigm
- **GPT-2 (2019, 1.5B Parameters)**: Scaled up model size, trained on web-scale data, Last publicly available weights
- **GPT-3 (2020, 175B Parameters)**: Enabled in-context learning, 100x boost, ~800GB file size
- **GPT-4 (2023, ~1T parameters (estimated))**: Multimodal, stronger reasoning and generalization

Applications (Wide range of NLP tasks through text generation)

- Text generation (e.g., chat, storytelling, code)
- Summarization
- Translation
- Question answering
- Semantic search and reasoning tasks

GPT's Architecture Deep Dive

How GPT Works

- **Multiple stacked decoder layers** with self-attention mechanisms
- **No encoders** - purely focused on text generation
- **Unidirectional processing** - considers only left-side context
- **Next word prediction** based on preceding words
- **High-quality text generation** from learned context patterns
- **Contextual understanding** combined with text creation capability

GPT Training Data

Pre-training Foundation

- **Massive, diverse datasets** including:
  - Web pages
  - Books and articles
  - Billions of words total
- **Different datasets** for each GPT version
- **Large-scale exposure** to language patterns and context
- **Quality varies** but emphasis on diversity and scale

Causal Language Modeling

GPT's Single Training Objective

- **Core task**: Predict the next word in a sequence
- **Method**: Uses context from preceding words only
- **Example**: "I love drinking fresh ___" → "lemonade"
- **Key benefits**:
  - Learns word relationships and context
  - Develops language patterns
  - Enables coherent text generation
- **Simplicity**: No masked language modeling or next sentence prediction

GPT's Ideal Use Cases

Where GPT Excels

- **Text Generation**: Story creation, creative writing, conversations
- **Instruction Following**: Responding to prompts and commands
- **Translation**: Converting text between languages
- **Summarization**: Creating concise summaries of longer texts
- **Code Generation**: Programming assistance and code completion
- **Versatile Applications**: Any task involving text output

Number of Parameters – GPT 3

Component	Parameter	Formula / Size	Total Parameters (Approx.)
Embedding Layer	Token Embeddings	Vocab × d_model = 50K × 12288	614.4M
	Positional Embeddings	n_ctx × d_model = 2048 × 12288	25.2M
Self-Attention (per layer)	Q, K, V, Output	4 × d_model × d_model = 4 × 12288^2	604.6M per layer
Feedforward (per layer)	2 layers (gelu)	d_model × d_ff + d_ff × d_model	1.2B per layer
LayerNorms (per layer)	Two per layer	2 × d_model	24.6K per layer
Total per layer	–	Self-attn + FFN + norms	≈ 1.8B per layer
Transformer Block Total	96 × 1.8B	–	≈ 172.8B
Final LayerNorm	–	d_model	12.3K
Output Layer (tied)	Shared with token embedding	–	— (tied with input embedding)
Total Parameters	–	Sum of above	**≈ 175B**

- **d_model = 12288**
- **n_layers = 96**
- **n_heads = 96 → each head has d_k = d_v = 128**
- **d_ff = 4 × d_model = 49152**
- **Vocabulary size ≈ 50,000**
- **Sequence length (n_ctx) = 2048**

GPT vs BERT Architecture

Aspect	GPT	BERT
Architecture	Decoder-only	Encoder-only
Text Processing	Unidirectional (left-to-right)	Bidirectional
Primary Goal	Text generation	Text understanding
Context	Previous words only	All surrounding words
Use Cases	Generation tasks	Classification/extraction